http://www.devx.com Printed from http://www.devx.com/Java/Article/22134/1954

Simplify Your Web App Development Using the Spring MVC Framework

Struts is in fairly widespread use in the Java world, but the Spring MVC framework promises to provide a simpler alternative to Struts for separating presentation layer and business logic. Learn how to build a simple stock trading Web application using Spring's MVC framework.

by Javid Jamae

In a previous article, I introduced you to the Spring framework, showed you how to use Spring's basic functionality to create objects, and how to do some simple database interactions. In this follow-up article I will introduce you to the Spring model-view- controller (MVC) framework by walking you through the creation of a simple stock-trading Web application.

MVC is a pattern that helps separate presentation from business logic. In short, in an MVC application all Web requests are handled by controllers. A "controller" is responsible for interpreting the user's request and interacting with the application's business objects in order to fulfill the request. These business objects are represented as the "model" part of the MVC. Based on the outcome of the request execution, the controller decides which "view" to forward the model to. The view uses the data in the model to create the presentation that is returned to the user.

If you've ever worked on a Web application, chances are that you've worked with either a custom MVC framework or an existing framework, such as Struts. Struts is in fairly widespread use in the Java world, but the Spring MVC framework promises to provide a simpler alternative to Struts.

In this article I will walk you through the development of a Web-based stock-trading application. This application will have three primary workflows each of which will cover a different type of controller available in Spring. After reading this article, you should have enough technical and business knowledge to build a full-fledged stock-trading application and compete head-on with existing discount brokerage firms. … OK, not really, but you will at least have enough knowledge to get you started on your own Web projects using Spring.

You can download the Spring distribution from http://www.springframework.org/download.html. If you want to see the application in action, download the source code for this article, which includes a deployable war file.

Author's Note: The code in this article has only been tested on Tomcat 5.5.

The Platform

There are many different Web application servers and development environments to choose from. I won't describe any single environment here because I want to focus the discussion on the Spring framework and not the underlying tools. However, I developed this sample application using Tomcat 5.5, Java 1.5, Spring 1.1, and Eclipse 3.0 using the Sysdeo Tomcat plugin. I have used Spring with other application servers and with different versions of Java 1.4.x without any big surprises, so you should be safe if you have a slightly different environment.

Getting Started

Before diving into Spring, set up a mechanism for deploying your application code into your Web server and set up an application context if necessary. In Tomcat 5.5, the application context automatically takes on the name of the war file (or expanded war directory) that you deploy. My war file is called "tradingapp.war" and my context is consequently "/tradingapp."

Here is the basic directory structure that you need to create for your Web app:

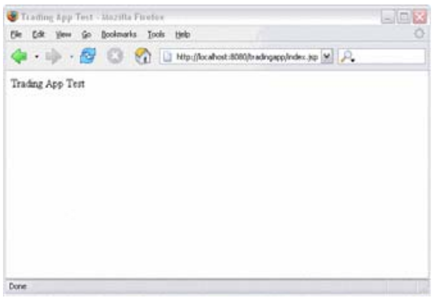
```
/WEB-INF
          /src              (java classes will go in here)
          /jsp              (application jsps will go in here)
          /lib              (jar files that our app depends on will go in here)
          /classes       (compiled class files will go in here)
```

In order to test out the application server setup, create a file called index.jsp (as follows) and put it in the root of your application directory:

```
/index.jsp
<html>
<head><title>Trading App Test</title></head>
<body>
Trading App Test
</body>
</html>
```

Try to pull up the page by going to http://localhost:8080/tradingapp/index.jsp (see Figure 1). (The port may vary depending on the application server you are using.)

On most application servers, users can access files in the root directory of the context. Hence, you were able to hit the index.jsp file directly. In an MVC architecture, you want the controller to handle all incoming requests. In order to do this, it is a good idea to keep your JSP files in a place where they are not directly accessible to the user. That is why we will put all of our application JSP files in the WEB-INF/jsp directory. We'll come back to this soon, but for now let's find out how to access a Spring controller.


Trading App Test - Mozilla Firefox

http://localhost:8080/tradingapp/index.jsp

Trading App Test

Done

Figure 1. Clear the Deck: Pull up the index.jsp page to make sure that you can retrieve a simple JSP page.

Accessing a Controller

Like in other Web frameworks, Spring uses a servlet called DispatcherServlet to route all incoming requests (this pattern is sometimes called the Front Controller pattern). This servlet should be defined in the Web deployment descriptor as shown here:

```
/WEB-INF/web.xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC '-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN'
 'http://java.sun.com/dtd/web-app_2_3.dtd'>

<web-app>
     <servlet>
          <servlet-name>tradingapp</servlet-name>
          <servlet-class>
               org.springframework.web.servlet.DispatcherServlet
          </servlet-class>
          <load-on-startup>1</load-on-startup>
     </servlet>

     <servlet-mapping>
          <servlet-name>tradingapp</servlet-name>
          <url-pattern>*.htm</url-pattern>
     </servlet-mapping>

     <welcome-file-list>
          <welcome-file>index.jsp</welcome-file>
     </welcome-file-list>

</web-app>
```

In the web.xml file I've defined a servlet mapping that forces any URL that ends in .htm to reroute to the tradingapp Servlet (the DispatcherServlet). This servlet analyzes the request URL and determines which controller to pass control on to by using a URL mapping defined in a Spring XML file. This Spring XML file must exist in the /WEB-INF directory and it must have the same name as the servlet name that you defined in the web.xml with a "-servlet.xml" appended to it. Thus, we will create a file in the /WEB-INF directory called "tradingapp-servlet.xml."

Here is the Spring XML file that the DispatcherServlet will use:

```
/WEB-INF/tradingapp-servlet.xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">

<beans>
     <bean id="portfolioController"
           class="com.devx.tradingapp.web.portfolio.PortfolioController">
     </bean>

     <!-- you can have more than one handler defined -->
     <bean id="urlMapping"
          class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
          <property name="urlMap">
               <map>
                    <entry key="/portfolio.htm">
                         <ref bean="portfolioController" />
                    </entry>
               </map>
          </property>
     </bean>
</beans>
```

I called the bean urlMapping, but Spring should pick up the mapping regardless of what you name it. In fact, you can have multiple handler-mapping objects defined. The SimpleUrlHandlerMapping class has a map property called urlMap that maps URLs to objects that implement the Controller interface. When the DispatcherServlet looks in this bean definition file, it will load the mapping and use it to determine which controller to route to. I've created a reference to the PortfolioController.

There are several different types of controllers available in the Spring framework. In this article, I will show you three different types:

1. For the portfolio page, I will create the simplest type of controller: one that implements the Controller interface directly.
2. Next I'll create a logon form that will use a SimpleFormController class.
3. Last, I'll create a trade wizard using the AbstractWizardFormController class.

The Portfolio Page

I first want to change the index.jsp page to redirect to my portfolio page by going through the PortfolioController. I'll also create a JSP called include.jsp that all of my JSP files can include in order to inherit a common set of properties and tag library definitions.

```
/index.jsp
<%@ include file="/WEB-INF/jsp/include.jsp" %>

<core:redirect url="/portfolio.htm"/>

/WEB-INF/jsp/include.jsp
<%@ page session="false"%>
```

Now I'll create a simple portfolio view and have my controller route to it just to make sure everything is working.

```
/WEB-INF/jsp/portfolio.jsp
<%@ include file="/WEB-INF/jsp/include.jsp" %>

<html>
<head><title>Portfolio Page</title></head>
<body>
Portfolio Page
</body>
</html>
```

Before writing the Controller, you should add a few jar files to your /WEB-INF/lib directory:

Jar	Description	Source
spring.jar	Main spring jar file	[spring-dist]/dist
log4j-1.2.8.jar	log4j logging package	[spring-dist]/lib/log4j
commons-logging.jar	Jakarta Commons logging package	[spring-dist]/lib/jakarta-commons
jstl.jar	Java standard tag library	[spring-dist]/lib/j2ee
standard.jar	Jakarta standard tag library	[spring-dist]/lib/jakarta-taglibs
taglibs-string.jar	Jakarta tag library used for String manipulation	http://jakarta.apache.org/taglibs/
commons-lang-2.0.jar	Jakarta Commons language API	http://jakarta.apache.org/commons
jfl.jar	Financial library used to get stock-quotes online	http://www.neuro-tech.net/archives/000032.html

Most of these jars are available in the Spring distribution. All of them are available with the downloadable code for this article.

Now create a class called PortfolioController:

```
/WEB-INF/src/com/devx/tradingapp/web/PortfolioController.java
package com.devx.tradingapp.web;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.Controller;

public class PortfolioController implements Controller {

    public ModelAndView handleRequest(HttpServletRequest request,
            HttpServletResponse response) {

        return new ModelAndView("/WEB-INF/jsp/portfolio.jsp");
    }

}
```

The Controller interface defines a single method signature:

```
public ModelAndView handleRequest(HttpServletRequest request,
HttpServletResponse response) throws java.lang.Exception;
```

The object that is returned by the handleRequest method is of the type ModelAndView. As you probably guessed, the ModelAndView object represents the Model and View in the MVC pattern. ModelAndView has several contructors. The one we're using right now just takes a string that represents the view that we want to forward to. Because our portfolio.jsp page doesn't have any dynamic content (yet), we don't need to return a model object.

Compile the PortfolioController and make sure the compiled file is under the WEB-INF/classes directory structure (i.e. /WEB-INF/classes/com/devx/tradingapp/web/PortfolioController.class). Now you can deploy your application and pull up the index.jsp page again. Go to http://localhost:8080/tradingapp/index.jsp and you should see what is shown in Figure 2.

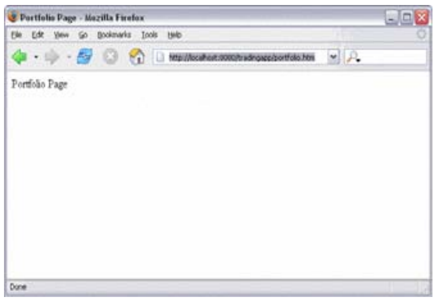

Portfolio Page - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://localhost:8080/tradingapp/portfolio.htm
Portfolio Page
Done

Figure 2. Back to the Portfolio: The index.jsp page should now redirect you to the Portfolio Page.

Now I'll make it a little easier to specify views from within the Controller objects. The goal is to avoid having to type out the full path to a JSP inside of my Controller code. I can achieve this through use of a ViewResolver that I'll define in my tradingapp-servlet.xml file. The ViewResolver appends a prefix and suffix to the view that the Controller returns.

```
/WEB-INF/tradingapp-servlet.xml
    <bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="viewClass"><value>org.springframework.web.servlet.view.JstlView</value></property>
        <property name="prefix"><value>/WEB-INF/jsp/</value></property>
        <property name="suffix"><value>.jsp</value></property>
    </bean>
```

Now I can simplify my PortfolioController:

```
/WEB-INF/src/com/devx/tradingapp/web/PortfolioController.java
package com.devx.tradingapp.web;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.Controller;

public class PortfolioController implements Controller {

    public ModelAndView handleRequest(HttpServletRequest request,
            HttpServletResponse response) {

        return new ModelAndView("portfolio");
    }

}
```

Just to make sure that the Portfolio Page still loads after this change, reload the page in your Web browser (you may have to reload the application context or restart your application server if your application server does not support hot-deploy functionality).

Now I want to make the portfolio page more interesting! First, I'll add some custom-tag library definitions to the include.jsp file. Using custom tags helps me keep my presentation logic separate from the presentation itself.

```
/WEB-INF/jsp/include.jsp
<%@ page session="false"%>

<%@ taglib prefix="core" uri="http://java.sun.com/jstl/core" %>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jstl/fmt" %>
<%@ taglib prefix="str" uri="http://jakarta.apache.org/taglibs/string-1.1" %>
```

Next, update your PortfolioController (see the code in Listing 1).

In the new version of the Controller, I've added a model. I use the three-argument constructor for ModelAndView that takes the view, the string name that the JSPs will use to refer to the model, and the model object itself. In Spring, a model is usually just a java.util.Map object. I put two elements on the model, a cash amount, and a list of portfolio line items. Each line item is of type PortfolioItemBean, a JavaBean that I created. This bean is primarily for View purposes and is created using data from the underlying business object called Portfolio. Listing 2 shows these classes.

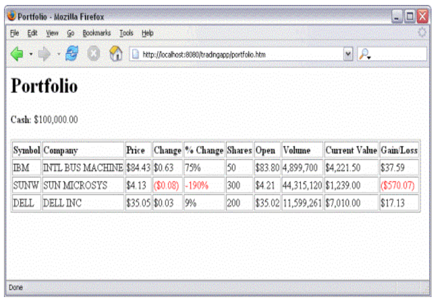
You'll also notice that I'm using a class called QuoteFactory to obtain a Quote object using a stock symbol. These classes are part of the Neurotech Java Financial Library, a simple, open source API that can be used to retrieve stock quotes from the Web as well as other simple financial tasks.

Listing 3 shows the updated tradingapp-servlet.xml file and the portfolio.jsp file. If all goes well, when you deploy and reload the page you should see something very similar to Figure 3.

The cool thing is that the PortfolioController retrieves quotes from the Web using the Java Financial Library, so if you keep refreshing the page during trading hours, you'll get updated quotes.

In summary, here is the flow to this page:

1. The user goes to the portfolio.htm page.
2. He is routed to the dispatcher Servlet (because all .htm pages are directed to the dispatcher Servlet).
3. The DispatcherServlet loads the tradingapp-servlet.xml file and routes the user to the PortfolioController.
4. The PortfolioController creates the model and view and passes control back to the DispatcherServlet.
5. The DispatcherServlet makes the model data accessible via session or request parameters.
6. The DispatcherServlet routes to the JSP pages.
7. The JSP is rendered and the presentation is sent back to the user.


Portfolio - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://localhost:8080/tradingapp/portfolio.htm
Done

Portfolio

Cash: $100,000.00

Symbol	Company	Price	Change	% Change	Shares	Open	Volume	Current Value	Gain/Loss
IBM	INTL BUS MACHINE	$84.43	$0.63	75%	50	$83.80	4,899,700	$4,221.50	$37.59
SUNW	SUN MICROSYS	$4.13	($0.08)	-190%	300	$4.21	44,315,120	$1,239.00	($570.07)
DELL	DELL INC	$35.05	$0.03	9%	200	$35.02	11,599,261	$7,010.00	$17.13

Figure 3. Improving the Portfolio: The portfolio view uses custom tags to display model data that is provided by the PortfolioController.

The Logon Page

Now you know how to create a page with dynamically updated content, but what good is a Web application that has no way to capture user input from an HTML form? We could implement the Controller interface in order to do form processing, but that would require us to do a lot of manual processing or request parameters and storing things on the session. Personally, I've hand coded more form-processing code in this fashion than I ever hope to again. Thus, out of sheer laziness, I will show you how Spring simplifies form handling via the SimpleFormController by building a logon page.

Listing 4 shows the logon JSP. The <spring:bind> tag and the <spring:hasBindErrors> tag are in the spring.tld tag library descriptor that comes with Spring. Add this file to your WEB-INF directory and then add the following to your web.xml and include.jsp files.

```
/WEB-INF/web.xml
     <taglib>
          <taglib-uri>/spring</taglib-uri>
          <taglib-location>/WEB-INF/spring.tld</taglib-location>
     </taglib>

/WEB-INF/jsp/include.jsp
<%@ page session="false"%>

<%@ taglib prefix="core" uri="http://java.sun.com/jstl/core" %>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jstl/fmt" %>
<%@ taglib prefix="str" uri="http://jakarta.apache.org/taglibs/string-1.1" %>
<%@ taglib prefix="spring" uri="/spring" %>
```

The bind tag tells Spring to bind the enclosed form element to the bean property specified by the path attribute. For example the username input parameter will be bound to the username field on a bean named "credentials." The credentials bean is a special type of bean called a "command" or a "form-backing bean." Spring uses command beans for data binding while processing forms. The name used to reference this bean and its class type is defined in the tradingapp-servlet.xml file in the logonForm bean (see Listing 5).

I've also added a new entry to the handler mapping to point to our logon form when the user navigates (or is routed) to the logon.htm URL. The logon form has several properties:

- commandClass—the class of the object that will be used to represent the data in this form.
- commandName—the name of the command object.
- sessionForm—if set to false, Spring uses a new bean instance (i.e. command object) per request, otherwise it will use the same bean instance for the duration of the session.
- validator—a class that implements Spring's Validator interface, used to validate data that is passed in from the form.
- formView—the JSP for the form, the user is sent here when the controller initially loads the form and when the form has been submitted with invalid data.
- successView—the JSP that the user is routed to if the form submits with no validation errors.

Here is the code for the Controller:

```
/WEB-INF/src/com/devx/tradingapp/web/LogonFormController.java
package com.devx.tradingapp.web;

import javax.servlet.ServletException;

import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.SimpleFormController;
import org.springframework.web.servlet.view.RedirectView;

public class LogonFormController extends SimpleFormController {
    public ModelAndView onSubmit(Object command) throws ServletException {
        return new ModelAndView(new RedirectView(getSuccessView()));
    }

}
```

There's really not much to it. All the FormController does is forward to the success view. The validator takes care of authenticating the user. If the user provides an invalid username or password, the validator will return the user to the form and display an error message. Listing 6 shows the code for the validator.

The validator has a supports method and a validate method. The supports method is called to see if the validator supports a given object type. We want our validator to be able to validate objects of the type Credentials, because that is the command object that we are using for this form.

The validate method does some checks and calls the rejectValue method on the Errors class if validation fails. This method takes four parameters: the name of the field that failed validation, the key value of the error message that exists in a resource file (I'll explain in a second), a string array of values to substitute into the error message in the resource file, and the default error message that should be displayed if the resource file cannot be found. If any errors are found on the Errors object after validate is done running, Spring will forward control back to the original form JSP so that the errors can be displayed.

You may have noticed above that I also added a ResourceBundleMessageSource bean definition to the tradingapp-servlet.xml file. This is a reference to a properties file that contains application messages that can be defined and accessed throughout the application. These messages can be accessed in many ways, including being directly accessed from JSPs or, as we just saw, by the Errors object to provide error messages. This file must exist in the top level of your classpath, so I just put it in the WEB-INF/classes directory. I've jumped the gun and included all the error messages that our application will use in the file:

```
/WEB-INF/classes/messages.properties
error.login.not-specified=User credentials not specified (try guest/guest).
error.login.invalid-user=Username not valid, try 'guest'
error.login.invalid-pass=Password not valid, try 'guest'
error.trade.insufficient-funds=You do not have enough money to place this order
error.trade.not-enough-shares=You do not have that many shares
error.trade.dont-own=You don't own this stock
error.trade.invalid-symbol=Invalid ticker symbol: {0}
```

Here is the code for the infamous Credentials class (the command/form-backing object):

```
WEB-INF/src/com/devx/tradingapp/business/Credentials.java
package com.devx.tradingapp.business;

public class Credentials {
    private String username;

    private String password;

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;

    }
}
```

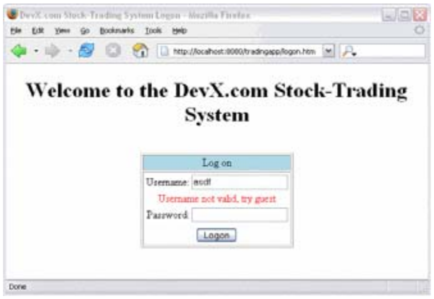
Now you can change the index.jsp page in the root directory to point to logon.htm.

```
/index.jsp
<%@ include file="/WEB-INF/jsp/include.jsp" %>

<core:redirect url="/logon.htm"/>
```

Now if you load index.jsp, you will see the logon page, as shown in Figure 4.

In summary, here is the flow through this form:

1. The user goes to the logon.htm page.
2. He is routed to the dispatcher servlet (because all .htm pages are directed to the dispatcher Servlet).
3. The DispatcherServlet loads the tradingapp-servlet.xml file and routes the user to the LogonFormController.
4. The LogonFormController loads the command bean (form-backing object) and routes the user to the page defined in the formView (logon.jsp).
5. The user fills out the form and submits it.
6. The user is directed back to the controller and in turn the validator.
7. If the validator fails, the user is sent back to the form view and error messages are displayed (back to logon.jsp and step 5).
8. If the validator doesn't fail, the controller's onSubmit method is called and the user is forwarded to the successView JSP (portfolio.jsp).


DevX.com Stock-Trading System Logon - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://localhost:8080/tradingapp/logon.htm
Log on
Logon
Done

Welcome to the DevX.com Stock-Trading System

Username: asdf

Username not valid, try guest

Password:

Figure 4. Validating: When the logon page does not validate correctly, the user is returned to the form page where the validation errors are displayed.

The Trade Wizard

Often times we have more than one screen that a user interacts with in order to complete a given task. This sequence of screens is often called a "wizard." The last MVC component I will introduce you to is the AbstractWizardFormController. This controller allows you to carry the same command object through an entire flow of Web pages, thus allowing you to break up the presentation into multiple stages that act on the same model.

For the example, I'll create a wizard that allows a user to trade stock. This wizard will start with a page that takes the order and then goes to a confirmation page where the user will choose to either execute or cancel the order. If the order is cancelled, the user will be returned to the portfolio page, if the order is executed the user is sent to an acknowledgement page to tell them that their order was filled.

First I need a way to get to the trade page, so I'll put a link to the trade page on the portfolio page.

```
/WEB-INF/jsp/portfolio.jsp
<br>
```

```
<a href="<core:url value="trade.htm"/>">Make a trade</a><br/>
<a href="<core:url value="logon.htm"/>">Log out</a>
<br>
</body>
</html>
```

Now I'll update the tradingapp-servlet.xml so that it knows what to do when the user clicks on the trade.htm link.

```
/WEB-INF/tradingapp-servlet.xml
     <bean id="tradeForm" class="com.devx.tradingapp.web. TradeFormController">
        <constructor-arg index="0">
            <ref bean="portfolio"/>
        </constructor-arg>
    </bean>

     <!-- you can have more than one handler defined -->
     <bean id="urlMapping"
          class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
          <property name="urlMap">
               <map>
                    <entry key="/portfolio.htm">
                         <ref bean="portfolioController" />
                    </entry>
                    <entry key="/logon.htm">
                         <ref bean="logonForm"/>
                    </entry>
                    <entry key="/trade.htm">
                         <ref bean="tradeForm"/>
                    </entry>
               </map>
          </property>
     </bean>
```

The TradeFormController should exist as /WEB-INF/src/com/devx/tradingapp/web/TradeFormController.java. But there is a lot to the TradeFormController, so I'll explain it in segments.

```
public class TradeFormController extends AbstractWizardFormController {

    private Portfolio portfolio;

    public TradeFormController(Portfolio portfolio) {
        this.portfolio = portfolio;
        setPages(new String[] { "trade", "trade-confirm" });
        setCommandName("trade");
    }
```

The TradeFormController extends AbstractWizardFormController. I pass in a Portfolio object so that the trade form knows what my buying limit is and can add and remove stock from the portfolio. The setPages method in the constructor tells the form the sequence of pages we are going to call.

The strings in this array relate to views that are resolved by the ViewResolver just like in the other controllers. These pages are indexed by Spring starting at 0. Thus the trade view is index 0 and the trade-confirm view is index 1. The indexes exist because we may have an event that causes us to skip or go back to a previous page. The setCommandName method sets the name of the command object that we are going to use for this form. This object will be created in the formBackingObject method.

```
    protected Object formBackingObject(HttpServletRequest request) {
        Trade trade = new Trade();
        trade.setBuySell(Trade.BUY);
        return trade;
    }
```

This method will create the command object and set it in its initial state. This method is called before the user is directed to the first page of the wizard. Subsequent submits will call several methods on the abstract controller. The main methods that are called are onBind, validatePage, and getTargetPage.

```
onBind
    protected void onBind(HttpServletRequest request, Object command,
            BindException errors) {

        Trade trade = (Trade) command;

        if (symbolIsInvalid(trade.getSymbol())) {
            errors.rejectValue("symbol", "error.trade.invalid-symbol",
                    new Object[] { trade.getSymbol() },
                    "Invalid ticker symbol.");
        } else {
            Quote quote = null;
            try {
                quote = new QuoteFactory().getQuote(trade.getSymbol());
            } catch (QuoteException e) {
                throw new RuntimeException(e);
            }
            trade.setPrice(quote.getValue());
            trade.setSymbol(trade.getSymbol().toUpperCase());
        }
    }
```

The onBind method is called before any validation occurs for each submit. I override the onBind method in order to get a quote on the symbol the user is trying to purchase and set the price and symbol on the command object. Keep in mind that even if onBind adds errors to the BindException, the validatePage method will still be called.

```
    protected void validatePage(Object command, Errors errors, int page) {
        Trade trade = (Trade) command;

        if (tradeIsBuy(trade)) {
            if (insufficientFunds(trade)) {
                errors.reject("error.trade.insufficient-funds",
                        "Insufficient funds.");
            }
        } else if (tradeIsSell(trade)) {
            if (portfolio.contains(trade.getSymbol()) == false) {
                errors.rejectValue("symbol", "error.trade.dont-own",
                        "You don't own this stock.");
            } else if (notEnoughShares(trade)) {
                errors.rejectValue("quantity", "error.trade.not-enough-shares",
                        "Not enough shares.");
            }
        }
    }
```

validatePage
The validatePage method is called after the onBind method. It acts in the same way as the validator did in the logon controller. The validatePage method is passed the page number for the page in the flow that you are validating. This can be used if you need to do custom validation for each page in the flow. If you wanted to, you could create validator objects for each page and use the validatePage method to delegate to the appropriate validator based on the page index that was passed in.

getTargetPage
The getTargetPage method will be called in order to find out which page to navigate to. This method can be overridden in your controller, but I use the default implementation, which looks for a request parameter starting with "_target" and ending with a number (e.g. "_target1"). The JSP pages should provide these request parameters so that the wizard knows which page to go to even when the user uses the Web-browser's back button.

The processFinish method is called if there is a submit that validates and contains the "_finish" request parameter. The processCancel method is called if there is a submit that contains the "_cancel" request parameter.

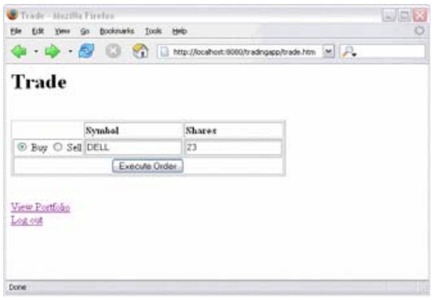
```
    protected ModelAndView processFinish(HttpServletRequest request,
            HttpServletResponse response, Object command, BindException errors) {
        Trade trade = (Trade) command;

        if (trade.isBuySell() == Trade.BUY) {
            portfolio.buyStock(trade.getSymbol(), trade.getShares(), trade
                    .getPrice());
        } else {
            portfolio.sellStock(trade.getSymbol(), trade.getShares(), trade
                    .getPrice());
        }
        return new ModelAndView("trade-acknowledge", "trade", trade);
    }

    protected ModelAndView processCancel(HttpServletRequest request,
            HttpServletResponse response, Object command, BindException errors) {
        return new ModelAndView(new RedirectView("portfolio.htm"));
    }
```

Listing 7 is the code for the Trade class (our command for this form). And now all you need are the JSP pages (see Listing 8 for the three JSP pages).

As you can see in the code, the Spring-specific request parameters are specified in the submit button name attribute. If all goes well, you should be able to pull up the first page of the trade wizard by going to http://localhost:8080/tradingapp/trade.htm (see Figure 5).

In summary, here is the flow through this form:

1. The user goes to the trade.htm page.
2. He is routed to the dispatcher servlet (because all .htm pages are directed to the dispatcher servlet).
3. The DispatcherServlet loads the tradingapp-servlet.xml file and routes the user to the TradeFormController.
4. The TradeFormController loads the command bean by calling the formBackingObject method and routes the user to the first page defined in the setPages call in the constructor (trade.jsp).
5. The user fills out the form and submits it.
6. The user is directed back to the controller, which parses the target page off of the request parameters from trade.jsp, binds and validates the command object, and forwards the user to the next page in the wizard.
7. If the validator fails, the user is sent back to the form view and error messages are displayed (back to trade.jsp).
8. If the validator doesn't fail, the controller forwards to the trade-confirm.jsp page.
9. The user submits an execute or cancel command, which will in turn tell the controller to execute either the processFinish or processCancel commands, respectively.
10. The user is forwarded to the page that the processFinish or processCancel command sends them to.


Trade - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://localhost:8080/tradingapp/trade.htm
Done

Trade

	Symbol	Shares
(•) Buy ( ) Sell	DELL	23
	Execute Order	

View Portfolio
Log out

Figure 5. Ready to Trade: The first page of the trade wizard allows you place an order for a stock trade.

I've gone over three different types of controllers provided by Spring, which should be enough to get you started on your own Web projects. The Spring framework contains quite a few more features than those I have gone over in this article and its predecessor. I have found Spring to be a very valuable tool in my software development toolbox, and I urge you to discover its vast utility for yourself.

Javid Jamae consults for Valtech, a global consulting group specializing in delivering advanced technology solutions. Valtech endeavors to help its customers through its global delivery model to create and/or maintain an affordable competitive advantage.